Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

A finite automaton learning system using genetic programming

H.H. Ehrenburg and H.A.N. van Maanen

Computer Science/Department of Algorithmics and Architecture

CS-R9458 1994

Report CS-R9458
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

A Finite Automaton Learning System using Genetic Programming

Herman Ehrenburg, Jeroen van Maanen
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
E-mail: {ehrenbur,jeroenm}@cwi.nl

Abstract

This report describes the Finite Automaton Learning System (FALS), an evolutionary system that is designed to find small digital circuits that duplicate the behavior of a given finite automaton. FALS is developed with the aim to get a better insight in learning systems. It is also targeted to become a general purpose automatic programming system.

The system is based on the genetic programming approach to *evolve* programs for tasks instead of explicitly programming them. A representation of digital circuits suitable for genetic programming is given as well as an extended crossover operator that alleviates the need to specify an upper bound for the number of states in advance.

AMS Subject Classification (1991): 68Q05, 68Q60, 68T05.
CR Subject Classification (1991): B.1.2, B.6.3, D.1.2, F.1.1, I.2.2, I.2.6.
Keywords & Phrases: Evolutionary computing, genetic programming, finite automata.
Note: Partially supported by the European Union through NeuroCOLT ESPRIT Working Group Nr. 8556, and by NWO through NFI Project ALADDIN under Contract number NF 62-376. This report is based on the Master's thesis of one of the authors [8]. Chapter 3 of this report was presented at the Benelearn'94 conference [9].

Table of Contents

1 Introduction **7**
1 Problems and objectives 7
2 Background 8

2 Evolutionary systems **9**
1 The structure of an evolution 10
2 Choices and variants 11
3 Theoretical Analysis 13
4 Genetic Algorithms 17
5 Genetic Programming 25

3 Finite Automaton Learning System **29**
1 The individual 30
2 Genetic operators 34
3 Circuit and architecture design 36

REFERENCES **37**

List of Figures

2.1 An evolution in an evolutionary system 10
2.2 The genetic operators in genetic algorithms 18
2.3 Example of an individual in genetic programming 26
2.4 The child operator in genetic programming 26

3.1 An individual in FALS 31
3.2 Example of a tree in a FALS individual 32
3.3 The child operator in FALS 33
3.4 Increasing the number of state variables 34
3.5 Decreasing the number of state variables 35

Chapter 1
Introduction

> As we succeed in broadening and deepening our knowledge—theoretical and empirical—about computers, we shall discover that in large part their behavior is governed by simple general laws, that what appeared as complexity in the computer program was, to a considerable extent, complexity of the environment to which the program was seeking to adapt its behavior.

(Herbert A. Simon, *The Science of the Artificial*, MIT Press, 1968).

1. Problems and objectives

Evolution can be seen as a procedure to develop solutions to optimization problems. A very useful feature of evolution in this context is that it does not seem to need a detailed analysis of the problem at hand. Just a method to judge the performance of a candidate solution – in Darwins terminology: the *fitness* of an individual [6] – is sufficient.

Valiant writes that "a program for performing a task has been acquired by *learning* if it has been acquired by any means other than explicit programming" [29]. If we can formulate a performance measure that tells us how good a program is at performing a certain task, we should be able to use it as a fitness criterion in a system that works analogous to natural evolution. This would give us one or more (near) optimal programs that perform the task. And, because no knowledge about the specific task is used, these programs can be viewed as acquired by learning. In other words, evolutionary systems can be thought of as learning algorithms.

The relevance of machine learning can be illustrated with the observation that programming todays computers is a very time consuming human activity and hardly ever results in very reliable programs. At present the development of programs, or software, can not keep pace

with the rate at which new machines are developed. As a result there is a shortage of programs. This shortage, usually referred to as the software crisis, constitutes a major problem. The solution to this problem might be to let a machine program itself. This leads to the question (attributed to Arthur Samuel in the 1950s):

> How can computers learn to solve without being explicitly programmed? In other words, how can computers be made to do what is needed to be done, without being told exactly how to do it?

We feel that learning algorithms that use the analogy with natural evolution can play an important role in solving the software crisis. We also hope that by studying these algorithms we get a better understanding of learning in general, *i.e.*, in the context of behavioral psychology, as well as evolution, *i.e.*, in the context of biology.

To guide our research we chose a practical problem to apply this method to. The problem we selected stems from the observation that every deterministic finite automaton can be represented as a digital circuit using only NAND gates and flip-flops. To find the *smallest* circuit given a finite automaton, however, is NP-complete. To study this problem in the context of evolutionary systems, we let circuits play the rôle of individuals. The fitness of a circuit will depend not only on the similarity of its behavior to a particular finite automaton, but also on the number of components in the circuit.

2. Background

FALS is an evolutionary system. Evolutionary systems are based on natural evolution as described by Darwin in *The Origin of Species* [6]. In natural evolution, individuals that survive produce offspring. So if a particular variety of individuals has a greater probability to survive than others, that variety produces more offspring, such that after a number of generations this variety will dominate the population. This principle can be used to implement optimization techniques by using the function to be optimized as a 'survival' criterion, *i.e.*, individuals that score higher are likelier to produce offspring. Optimization algorithms based on this principle are called evolutionary systems.

Two examples of evolutionary systems are genetic algorithms and genetic programming. Genetic algorithms, introduced by Holland [11], borrow not only the concept of evolution from biology, but also the idea of representing an individual by a code. This code is first used to construct the individual itself, in order to determine its fitness. In the reproductive phase of the system, two individuals are selected from the population on the basis of their fitness and their *codes* are combined to produce the codes of the offspring. Genetic programming is a technique that was described by Koza [15]. This method is very similar to genetic algorithms, but it applies to individuals that are represented as programs, *i.e.*, expression trees. Using this representation it is much easier to design systems that solve specific optimization problems.

We used this approach to design a representation of digital circuits as expression trees. Finite automata can be represented by fairly simple digital circuits, consisting of only NAND gates and flip-flops [16].

Chapter 2
Evolutionary systems

To give a framework for the finite automaton learning system that we are going to discuss in the next chapter, this chapter introduces the general idea of evolutionary systems as well as two popular classes of evolutionary systems: genetic algorithms and genetic programming. The description of genetic algorithms and genetic programming differs considerably from the ones given by Holland [11] and Koza [15] respectively. This is done for reasons of brevity and to smooth the transition to FALS.

An evolutionary system is a tool that can be used to solve optimization problems. In this chapter, we will assume that the optimization problem for evolutionary systems is to find a mathematical function, that fits a given set of $\langle \text{input}, \text{output} \rangle$-pairs. There are other optimization problems, than finding a function, that can be solved by evolutionary systems, but for our purposes it suffices to confine ourselves to mathematical functions.

Evolutionary systems are based on evolution in nature as described by Darwin in [6]. (The basis of this process is discrete genetics in accordance with so called Mendelian Laws.) In nature a species is able to adapt to an environment in a process called natural selection. Natural selection takes place in a so called evolution, which can be viewed as the repetition of two steps, namely survival and reproduction. Survival depends on the ability of the members of a species to stay alive long enough to reproduce. In the process of reproduction genetic material can be recombined which may result in offspring that is better adapted to the environment. The basic idea of evolutionary systems is to use the mechanism of natural selection to find a solution to an optimization problem. To do this a set of candidate solutions suitably called a population of individuals forming a genetic pool using discrete genes satisfying Mendelian Laws is evolved until a satisfying solution to the optimization problem at hand is found.

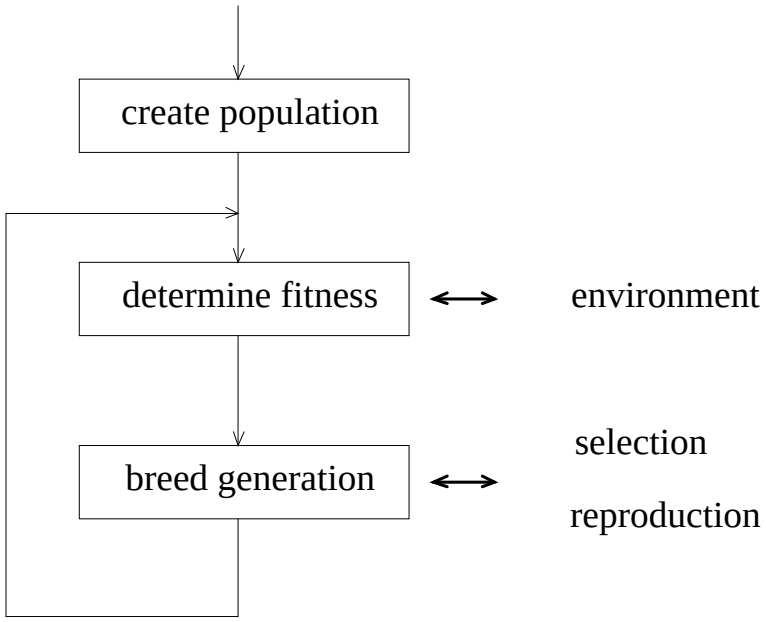

Figure 2.1: An evolution in an evolutionary system

1. The structure of an evolution

Figure 2.1 schematically represents the idea of an evolution in an evolutionary system. An evolution starts with the creation of a population called the first generation. The creation of the population is followed by the actual evolution which consists of the repetition of two steps called 'determine fitness' and 'breed generation' for a number of populations called generations.

The first step of a generation is to determine the fitness of each individual in the present generation. This is done by testing each individual with respect to fitness criteria (the environment). Fitness is a measure that expresses how well an individual is adapted to the environment and is the evolutionary system equivalent of the *probability* that a member of a species in nature will survive and reproduce.

The second step of a generation is to breed the population that will become the next generation. A population is bred by breeding a number of individuals. Breeding of an individual is done in two steps, called selection and reproduction. In the selection step one or two individuals are selected. Selection is biased in favor of fitter individuals and corresponds to survival in nature till the time of reproduction. In the reproduction step a new individual is produced based on the selected individuals. The evolution continues until an optimally fit individual is found, *i.e.*, a solution to the optimization problem is found, or some other termination criterion is satisfied.

2. Choices and variants

The first generation in an evolution can be randomly generated or it can be supplied by the user of the evolutionary system. The optimal way to construct the first generation depends among other things upon the way individuals are represented, which in turn depends on the kind of evolutionary system.

We restricted ourselves to environments that can be represented by sets of $\langle \text{input}, \text{output} \rangle$-pairs. An environment expresses an optimization problem, namely to find a mathematical function that fits the set of $\langle \text{input}, \text{output} \rangle$-pairs as well as possible. In order to make sure that there is a function that fits the $\langle \text{input}, \text{output} \rangle$-pairs, we require that if two pairs have identical input components, they also have identical output components.

The fitness of an individual is expressed by some form of distance measure with respect to the solution of the problem to be optimized. It is determined by comparing the output of the individual to the environment. The fitness of an individual is used to determine its probability of being selected for reproduction in the breeding phase. For an evolutionary system to work, the probability of selection should be monotonous in the fitness, *i.e.*, the larger the fitness of an individual, the likelier it will be selected. There can also be secondary fitness criteria like the size of an individual. Such criteria can be used to punish the use of (a lot of) resources.

Selection can be done in several ways. We will discuss two, namely *fitness proportionate selection* and *tournament selection*. Fitness proportionate selection selects an individual from the population based on a probability distribution over the individuals in the population, where the probability of an individual to be selected is equal to its fitness value divided by the sum of the fitness values of the whole population. Tournament selection draws two individuals from the uniformly distributed population, so each individual has equal probability of being drawn. The fitness values of the two individuals are compared and the individual having the higher fitness value is selected. In some implementations of evolutionary systems larger tournaments are used. An important difference between fitness proportionate selection and tournament selection is, that for tournament selection it is not necessary to represent the fitness of an individual as an actual number. It is sufficient to compare the performance of the two individuals selected for the tournament with respect to the environment to determine which one is the fitter. Fitness proportionate selection requires a representation of fitness in real numbers or a representation that can be mapped onto the real numbers. Another advantage of tournament selection, apart from being less explicit in its representation of fitness, is that it takes less computational effort, a simple comparison operation for each individual to be selected suffices. Fitness proportionate selection requires the computation of the sum of the fitness values and for each selection a comparatively elaborate computation in order to determine which individual to select. For theoretical results obtained using fitness proportionate selection see [11] and Section 4.1. In the following we will assume that tournament selection is used.

Reproduction is the process of producing a new individual for the next generation. Reproduction is done by applying genetic operators to one or two individuals selected with tournament selection. There are several genetic operators. The most important ones are child, clone and mutate. The child operator takes two selected individuals and combines them to construct an individual called a 'child'. The two selected individuals are called 'par-

ents' or 'father' and 'mother'. The clone operator takes one selected individual and makes an exact copy of it called a 'clone'. The mutate operator takes one selected individual and makes a copy of it of which a part is randomly changed, the result is a 'mutant'. After a new individual is produced by reproduction it is added to the next generation. Note that the selected individuals are not part of the new generation. For an elaborate description of genetic operators in genetic programming, see [15, p. 99-112].

There are many possible termination conditions for an evolution. The most important one is of course the recognition of a satisfactory solution to the optimization problem. Because it is not guaranteed that a satisfying solution will be found there should also be other termination conditions. Usually there is a limit on the maximum number of generations in an evolution. Another termination criterion is the detection of a stable population, *i.e.*, a population that always breeds an identical population.

In this paper we assume that a population consists of a fixed number of individuals, the population size. A typical minimal population size required in order to get good results in an evolution is something like 500 individuals (see [15]). In general the bigger the population the higher the probability of a successful evolution, but the greater the effort it takes to compute each generation. This leads to trade-offs between approximation and running time of the process, which will be object of further study. As a consequence the population size is determined by the memory size and processing power of the machine on which the evolutionary system is implemented.

Evolutionary systems are well suited for parallel execution. A number of evolutions can be executed independently of each other, to increase the probability of finding an optimal fit individual in an evolution within a relatively small number of generations. During a generation determination of fitness can be done for all individuals in parallel as well. Reproduction during a generation can also be done completely in parallel for each new individual to be bred in producing the next generation. The order of execution of the generations is the only thing in evolutionary systems that has to be done sequentially. Since both steps in each generation can be done in parallel for all individuals in the population, a completely parallel implementation of an evolutionary system would speed up the execution with a multiplicative factor, of at least the population size. Possibly defying Amdahl's Law, which says that m processes working in parallel can only speed up a process by an $o(m)$ multiplicative factor.

In this chapter we restricted the environment to a set of $\langle \text{input}, \text{output} \rangle$-pairs. These pairs can also be compared in parallel to the individual. In general, determining fitness per individual can not be done in parallel. As we shall see in the case of FALS, some environments have to be compared sequentially to the individual to determine fitness.

Although evolutionary systems have been around for almost two decades, they have hardly been used until recently. The major problem of evolutionary system implementations is that they require computers with large memory and powerful processors. A large memory and powerful processors are needed to process as many individuals as possible per generation. The advances in memory size and processing power of sequential computers has only relatively recently made modest evolutionary systems worth implementing for real world applications. Implementation on parallel architectures hold great promise for the future of evolutionary systems.

3. Theoretical Analysis

Rabinovich, Sinclair and Wigderson analyzed symmetric quadratic dynamical systems [22]. It appears that these systems are very well suited to model evolutionary systems and in particular genetic variants of evolutionary systems. In quadratic dynamical systems outcomes only depend on pairwise collisions. Evolutionary Systems usually are quadratic dynamical systems, since in most evolutionary systems outcomes only depend on pairwise collisions. Below we give a description of some theoretical results obtained for quadratic dynamical systems in terms of evolutionary systems. The main result is that for (discrete-time, finite-dimensional) symmetric quadratic dynamical systems every trajectory that does not approach the boundary of the simplex converges to a fixed point. In terms of quadratic evolutionary systems: every population that does not loose variety over the generations of an evolution will end up as a stable population.

Take a finite set $\mathcal{N}$ of types of individuals and a population p which is a probability distribution $p = (p_i)$ over $\mathcal{N}$. To denote a second population we will also use $q = (q_i)$. In evolutionary systems two individuals are of the same type if they have the same representation. Normally a population is a multi-set over the types, but in this analysis we assume an infinite number of individuals and simplify this further to a probability distribution over the types. Let $\mathcal{F}$ be an operator on a population p that gives the next generation $\mathcal{F}(p)$. We will restrict ourselves to functions that model systems where the next population is formed by pair-wise interaction between individuals that carries over to a pair-wise operation on types, *e.g.*, a simple genetic operator like crossover takes two individuals and recombines them into two children. Such a function can be fully described by parameters β_{ijkl} that specify the probability that a particular pair of parents with types i and j will produce offspring with types k and l. We require the function to be symmetric, locally reversible and a-periodic. This is reflected in the following conditions on the parameters.

$$\beta_{ijkl} = \beta_{jikl} = \beta_{ijlk}$$
$$\beta_{ijkl} = \beta_{klij}$$
$$\beta_{ijij} > 0$$

Given these parameters β_{ijkl} we can define $\mathcal{F}(p)$ as a population q, where the probability of type q_l is the sum of the probabilities that l and k are the result of a crossover between i and j for all possible parents i, j and all possible siblings k.

Definition 1

We define a multiplication operator $\times$ on populations such that for every pair of populations p, q and every $l \in \mathcal{N}$

$$(p \times q)_l := \sum_{i,j,k \in \mathcal{N}} p_i q_j \beta_{ijkl}$$

The function $\mathcal{F}$ is now defined as

$$\mathcal{F}(p) := p \times p$$

An evolution is a time sequence $p(0), p(1), p(2), \ldots$, where $p(t)$ is generation t and $p(t+1) = \mathcal{F}(p(t))$. If the limit of this sequence exists, for some initial population $p(0)$, then, since $\mathcal{F}$ is continuous, the limit point $\pi = \lim_{t\to\infty} p(t)$ is a fixed point π of $\mathcal{F}$, *i.e.*, $\mathcal{F}(\pi) = \pi$. In order to prove that for every evolution and for every initial population the limit of the evolution exists, we introduce the entropy of a population.

Definition 2
The entropy of a population p is

$$H(p) := -\sum_i p_i \log p_i$$

Theorem 1
Entropy is strictly increasing in an evolution of a non-stable population.

Proof: The theorem follows if by each application of $\mathcal{F}$ in an evolution of a non-stable population entropy strictly increases. We will work with probability distributions on the set $\mathcal{N}^2$ of ordered pairs of types $p_i p_j$. Let p be a non-stable population and q a population such that $q = \mathcal{F}(p)$ after an application of $\mathcal{F}$. The function $\mathcal{F}$ can be decomposed into two steps. We will refer to these steps as birth and split. Denote $p_i p_j$ as p^2_{ij} for all $i, j \in \mathcal{N}$, then birth is a linear operation on the resulting pair distribution p^2. Split is a re-computation of the singleton probabilities.

We define the first step, birth, as a linear operator B by specifying that for all $k, l \in \mathcal{N}$

$$(B(p^2))_{kl} := \sum_{i,j\in\mathcal{N}^2} p^2_{ij}\beta_{ijkl}$$

So B is a matrix where both rows and columns correspond to pairs of types such that $B_{ij,kl} = \beta_{ijkl}$ for all $i, j, k, l \in \mathcal{N}$. Birth can be viewed as the transition in a Markov chain, where B is the transition matrix of this Markov chain. Now partition the space $\mathcal{N}^2$ according to the equivalence relation $\sim$ defined as $ij \sim kl$ if and only if $B^n_{ij,kl} > 0$ for some $n \in \mathbb{N}$. Then the restriction of B to each equivalence class defines an independent Markov chain, which is symmetric and a-periodic and thus has the uniform distribution as its unique stationary distribution. It is well known that this kind of Markov chain causes the entropy of any non-uniform probability distribution (in this case a non-stable population) to increase strictly on every step. Therefore birth gives a strict increase in the entropy of the pair distribution. By rewriting the definition of the entropy of p^2 it is easy to see that $H(p^2) = 2H(p)$ for every distribution p. Together with the next step this will enable us to conclude that the entropy of the population increases strictly with every generation provided the population is not stable.

The second step, split, is defined as follows

$$q_l := \sum_{k\in\mathcal{N}} (B(p^2))_{kl}$$

Split makes the pair of children k, l independent. The distribution q^2_{kl} over the children k, l has maximum entropy among all distributions on $\mathcal{N}^2$ with the marginals q_l, so entropy is increasing in both steps. ■

Define the variation distance between two populations p and q as

$$||p - q|| := \tfrac{1}{2} \sum_{i \in \mathcal{N}} |p_i - q_i|$$

It can be shown that

$$||p - q|| = \max_{A \subseteq \mathcal{N}} |p(A) - q(A)|$$

Write $p(t) \to \pi$ to denote that $\lim_{t \to \infty} ||p(t) - \pi|| = 0$.

Theorem 2
For every symmetric, a-periodic operator β and every initial population $p(0)$, there exists a stable population π such that $p(t) \to \pi$.

Proof: (sketch): Because entropy strictly increases for non-stable populations (see Theorem 1) and entropy of the population is bounded by $\log |\mathcal{N}|$, entropy must tend to some finite limit in an evolution. It can be shown by partitioning $\mathcal{N}$ into equivalence classes using the relation $\sim$ as defined in the proof above, that there is a function $\delta = \delta(\varepsilon)$ such that

$$\max \left\{ \left| p_{ij}^2 - p_{kl}^2 \right| \middle| \beta_{ijkl} > 0 \right\} \geq \varepsilon$$

implies that $H(\mathcal{F}(p)) - H(p) \geq \delta(\varepsilon)$. Therefore if $H(p(t))$ tends to a limit then $p(t)$ converges too. ■

Theorem 3
A population is stable if $p_i p_j = p_k p_l$ for all i, j, k, l such that $\beta_{ijkl} > 0$.

Proof: Recall from the proof of Theorem 1 that the condition $p_i p_j = p_k p_l$ for all i, j, k, l such that $\beta_{ijkl} > 0$, is precisely the condition that the pair distribution p_{ij}^2 is a stable distribution for the Markov chain induced by birth. This condition is equivalent to stability of p, since p changes only if p_{ij}^2 changes by application of B. To see this, note that split has no effect if birth has no effect and that split can not undo a change by birth because it causes entropy to increase. ■

We can conclude from Theorem 3 that in general there will be a continuum of stable populations for a particular β. In order to determine for what initial populations the evolution will converge to a particular limit we will now study a class of functions inv_π that assign values to populations, which are invariant under $\mathcal{F}$. We restrict our investigations to populations that have full support. A population has full support if $p_i > 0$ for all $i \in \mathcal{N}$. In a sense this is a pity, because together with the fact that we disregard mutation in this analysis, this excludes populations that have less than N individuals.

Theorem 4
Let π be any stable population with full support, then the function

$$\text{inv}_\pi(p) := \sum_{l \in \mathcal{N}} p_l \log \pi_l$$

is invariant under $\mathcal{F}$.

Proof: For each $l \in \mathcal{N}$ define the function $\Delta p_l(t) := p_l(t+1) - p_l(t)$. Expand Δp_l as

$$\Delta p_l(t) = \sum_{ijk} \left(p_{ij}^2(t) - p_{kl}^2(t) \right) \beta_{ijkl} \tag{3.1}$$

where we have used Definition 1 to rewrite $p_l(t+1)$ and symmetry of $\mathcal{F}$ to rewrite $p_l(t)$. Let $\alpha_l := \log \pi_l$, and note that, since π is stable, it follows from Theorem 3 that for all $i, j, k, l \in \mathcal{N}$ such that $\beta_{ijkl} > 0$

$$\alpha_i + \alpha_j = \alpha_k + \alpha_l \tag{3.2}$$

Using Equation 3.1 we may write

$$\begin{aligned} \mathrm{inv}_\pi(\mathcal{F}(p)) - \mathrm{inv}_\pi(p) &= \sum_{ijkl} \alpha_l (p_{ij}^2 - p_{kl}^2) \beta_{ijkl} \\ &= \tfrac{1}{4} \sum_{ijkl} (\alpha_k + \alpha_l - \alpha_i - \alpha_j)(p_{ij}^2 - p_{kl}^2) \beta_{ijkl} \end{aligned}$$

where we have used the symmetry properties of β. But Equation 3.2 implies that every term in this sum is zero, so inv_π is indeed invariant. ■

These invariants can be used to characterize the populations that will evolve to the same limit as follows.

Theorem 5
Suppose $p(t) \to p^*$ and $q(t) \to q^*$, where p^* and q^* have full support. If $\mathrm{inv}_\pi(p(0)) = \mathrm{inv}_\pi(q(0))$ for all invariants inv_π, then $p^* = q^*$.

Proof: We know in particular that $\mathrm{inv}_{p^*}(p(0)) = \mathrm{inv}_{p^*}(q(0))$ and hence by continuity of the operator $\mathcal{F}$ that $\mathrm{inv}_{p^*}(p^*) = \mathrm{inv}_{p^*}(q^*)$. Similarly, $\mathrm{inv}_{q^*}(p^*) = \mathrm{inv}_{q^*}(q^*)$. Writing out these equalities yields

$$\sum_l p_l^* \log p_l^* = \sum_l q_l^* \log p_l^*$$

$$\sum_l p_l^* \log q_l^* = \sum_l q_l^* \log q_l^*$$

which can be subtracted to give

$$\sum_l p_l^* \log \frac{p_l^*}{q_l^*} = \sum_l q_l^* \log \frac{p_l^*}{q_l^*}$$

But by Jensen's inequality applied to the convex function $x \log x$, the left-hand side of this equation is always non-negative and the right-hand side is always non-positive. Hence both sides must in fact equal zero, which happens if and only if $p^* = q^*$. ■

Theorem 5 gives us a finite system of equations whose *unique* solution is the limit point of the evolution of $p(0)$. These equations consist of the equalities $p_{ij}^2 = p_{kl}^2$ from Theorem 3 together with a finite base for the set of linear equations $\mathrm{inv}_\pi(p) = c_\pi$ where $c_\pi = \mathrm{inv}_\pi(p(0))$, this gives an equation for every stable population with full support π and every population with full support $p(0)$.

4. Genetic Algorithms

In genetic algorithms as described by Holland [11] an individual is a bit string representing a function which is a possible solution to an optimization problem. The genetic operators child, clone and mutate in genetic algorithms are illustrated in Figure 2.2. A child is constructed from two selected individuals called father and mother by taking a prefix of random length from the father and the complementary suffix from the mother. Which selected individual is father and which one is mother is determined at random. Although this is a very common way to construct children in genetic algorithms, it is not the only possibility. Often the number of cutting points is varied. That is, the bit strings of the parents are cut at several corresponding points and a child is created choosing each segment randomly from both parents. This gives a whole range of crossover operators starting with the one-point crossover all the way up to what is called uniform crossover, where every bit is selected at random from one of the parents. The latter form of crossover is assumed in the analysis in the next section. A clone is just a copy of a selected individual and a mutant is a copy of a selected individual, with one or more mutations. A mutation is a random change in the individual, *i.e.*, a randomly chosen bit is flipped.

Solving an optimization problem by means of genetic algorithms involves solving two other problems. Namely choosing the operators that can be part of a function and deciding how to represent a function in the form of a bit string. The operators that can be part of a function are elements of the so called operator set. Choosing a suitable operator set is done on the basis of the problem at hand and requires some knowledge of the kind of solution to the problem. How to decide on the representation of a function by a bit string is extensively discussed in [15, p. 63–72]. Having to solve these two problems is a major drawback of genetic algorithms. However despite this drawback genetic algorithms are a good method for optimization. In [11] Holland gave a theoretical basis for the use of genetic algorithms.

4.1 Analysis of a simple genetic algorithm

In [23] Rabinovich and Wigderson present an analysis of a simple genetic algorithm. Below we give the main results. We start by defining populations, fitness, selection and reproduction using the same framework as Section 3. Then we observe that with the assumption of some symmetry conditions on populations, we can represent populations much shorter than by listing all possible probabilities. Using this a lower bound and an upper bound on convergence to a population with maximal average fitness are given. Finally we look at populations that are stable under the reproduction operator.

Starting point for our definitions will be the notation used in Section 3. The fact that individuals are represented as bit strings in genetic algorithms suggests that we take $\mathcal{N} := \{0,1\}^n$ for some fixed $n \in \mathbb{N}$. Denote $|\mathcal{N}| = 2^n$ as N. For $i \in \{1,\ldots,N\}$ we identify i with the unique bit string x in $\mathcal{N}$ that satisfies

$$\sum_{j=0}^{n-1} x_j \cdot 2^j = i - 1$$

As in Section 3 populations p, q will be functions that assign probabilities to elements of $\mathcal{N}$. For this analysis we will assume the fitness $f(i)$ of a binary string i to be equal to the number

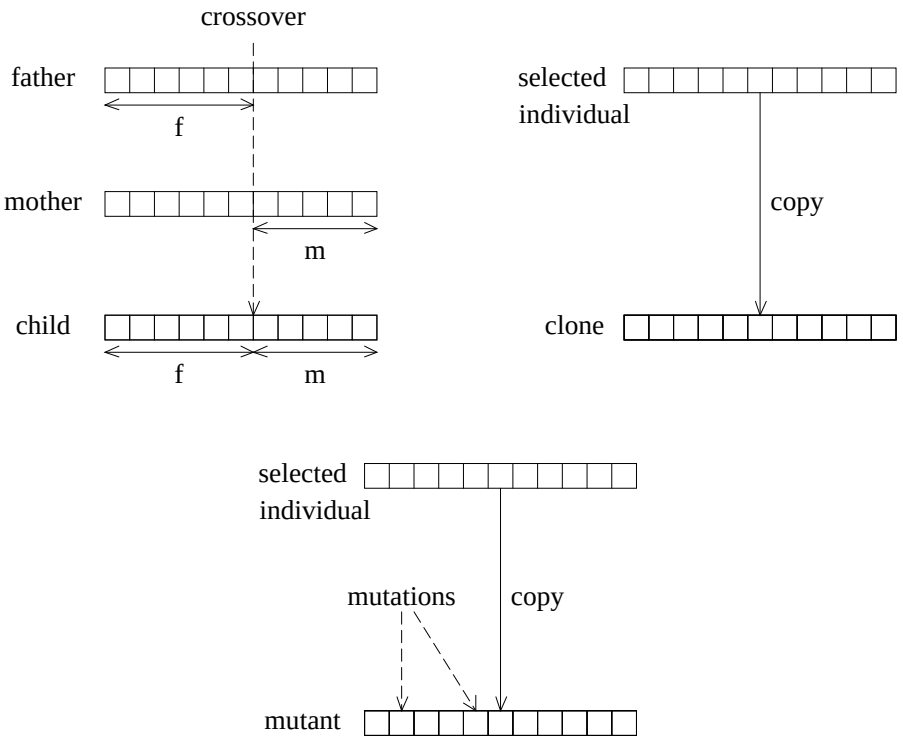

Figure 2.2: The genetic operators in genetic algorithms

of 1's it contains, formally $f(i) = \sum_{j=0}^{n-1} i_j$. The fitness of a population is expressed by the average fitness

$$\mathbf{Av}(p) := \sum_{i=1}^{N} p_i f(i)$$

We restrict populations to symmetric populations. Symmetric populations are populations that remain unchanged under bit permutation. So in a symmetric population, permutations of binary strings have equal probability. Define two operators on populations, that preserve symmetry: fitness proportionate selection W (weigh) and reproduction M (mate). Fitness proportionate selection changes the probability of a binary string proportional to its fitness.

Definition 3
Selection W is defined by $W(p) = q$, where the q_i are defined by

$$q_i = \frac{p_i}{\mathbf{Av}(p)} f(i)$$

Reproduction uses crossover on binary strings. Crossover combines two binary strings j and k to a binary string i. Crossover can be thought of as an operation in which each bit in i is selected with equal probability from the two bits with corresponding position index in j and k. This is the uniform variant of crossover. The probability that i will be the result of an application of crossover on j and k is denoted by $\mathbf{P}[i = j \times k]$, it is equal to the number of different ways in which i can be composed by crossover from j and k, divided by the total number of crossover compositions 2^n. This leads to the following definition of the new probabilities of binary strings after application of reproduction to p.

Definition 4
Reproduction M is defined by $M(p) = q$, where the q_i are defined by

$$q_i = \sum_{j,k \in \mathcal{N}} p_{jk}^2 \mathbf{P}[i = j \times k] \tag{4.3}$$

Note that we used the same notation for the pair-distribution p^2 as in Section 3. We now define $\mathcal{F}$ to be $M \circ W$, *i.e.*, $\mathcal{F}(p) := M(W(p))$. Application of this operator iteratively starting with an initial population $p(0)$ will like before give us a sequence of generations $p(0), p(1), p(2), \ldots$ that we will call an evolution. The fact that we restricted the analysis to symmetric populations combined with simplicity of the fitness function we chose to analyze enables us to represent populations as vectors much shorter than N. A way to do this is as follows.

Definition 5
Let p be a symmetric population over $\mathcal{N}$. Define the *histogram representation* a^p of p as the vector of length $n+1$ such that, if X is a random variable over $\mathcal{N}$ distributed according to p, for all $j \in \{0, \ldots, n\}$

$$A_j := \{\, i \in \mathcal{N} \mid f(i) = j \,\}$$
$$a_j^p := \mathbf{P}[X \in A_j]$$

Note that, because $\sum_{j=0}^{n} a_j^p = 1$ we can drop one of the elements of the vector and still be able to recompute the probabilities of all the types in the population. We get a slightly more complicated way to represent a populations by looking at probabilities of prefixes of 1's of varying lengths.

Definition 6
Let p be a symmetric population over $\mathcal{N}$. Define the *vector representation* e^p of p as the vector of length n such that, if X is a random variable over $\mathcal{N}$ distributed according to p, for all $j \in \{1, \ldots, n\}$

$$e_j^p := \mathbf{P}\left[\bigwedge_{k=0}^{j-1}(X_k = 1)\right]$$

where X_k is the kth bit of X.

So e_j is the probability that a random individual from population p has a prefix of at least j 1's. If the population is clear from the context we will drop the superscript from e or a. The histogram and vector representations a and e of a population p are related in the following way. For every $k \in \{1, \ldots, n\}$

$$\begin{aligned} e_k &= \mathbf{P}\left[\bigwedge_{i=0}^{k-1}(X_i = 1)\right] \\ &= \sum_{j=0}^{n} \mathbf{P}\left[\bigwedge_{i=0}^{k-1}(X_i = 1) \;\middle|\; X \in A_j\right] \cdot \mathbf{P}[X \in A_j] \\ &= \sum_{j=k}^{n} \frac{\binom{n-k}{j-k}}{\binom{n}{j}} a_j \\ &= \frac{1}{\binom{n}{k}} \sum_{j=k}^{n} \binom{j}{k} a_j \end{aligned}$$

As a consequence this means that

$$\begin{aligned} e_1 &= \frac{1}{n} \sum_{j=1}^{n} j a_j \\ &= \mathbf{Av}(p)/n \end{aligned}$$

The vector representation can be used to show that the average fitness of a population is not changed by applying the reproduction operator.

Theorem 6
For every symmetric population p

$$\mathbf{Av}(p) = \mathbf{Av}(M(p))$$

Proof: Let e be the vector representation of population p. Because $e_1 = \mathbf{Av}(p)/n$ the theorem follows if we prove that the expectation of a prefix of length 1 is preserved under M, *i.e.*, that $e_1^{M(p)} = e_1^p$. The probability of a prefix of length 1 after reproduction is equal to the sum of two probabilities, namely the probability that the string is the result of a crossover of two strings starting with a 1 and the probability that the string is the result of a crossover of a string starting with a 1 and a string starting with a 0. Thus

$$\begin{aligned} e_1^{M(p)} &= (e_1^p)^2 + 2\left(\tfrac{1}{2} e_1^p (1 - e_1^p)\right) \\ &= e_1^p \end{aligned}$$

So reproduction does not change the average fitness. ■

Writing out the probabilities as in the previous proof we can also show that $e_2^{W(p)} = (e_2^p + (e_1^p)^2)/2$. Of course the selection operator W *does* change the average fitness of the population. We will now proceed to express W in terms of the vector representation of a population. This will give us the change in e_1, which will enable us to express the change in average fitness after an application of W.

Proposition 7
Let p be an arbitrary population with vector representation e. If $p' = W(p)$, the kth coordinate of the vector representation e' of p' is given by

$$e'_k = \frac{n-k}{n} \cdot \frac{e_{k+1}}{e_1} + \frac{k}{n} \cdot \frac{e_k}{e_1}$$

Proof: We prove this by writing out e'_k in terms of the histogram representations a and a' of p and p'.

$$\begin{aligned} e'_k &= \frac{1}{\binom{n}{k}} \sum_{j=k}^{n} \binom{j}{k} a'_j \\ &= \frac{1}{\binom{n}{k}} \sum_{j=k}^{n} \binom{j}{k} \frac{j\, a_j}{n\, e_1} \\ &= \frac{1}{n\, e_1} \left(\frac{k}{\binom{n}{k}} \sum_{j=k}^{n} \binom{j}{k} a_j + \frac{k+1}{\binom{n}{k}} \sum_{j=k+1}^{n} \binom{j}{k+1} a_j \right) \\ &= \frac{1}{n\, e_1} \left(k\, e_k + \frac{n-k}{\binom{n}{k+1}} \sum_{j=k+1}^{n} \binom{j}{k+1} a_j \right) \\ &= \frac{1}{n\, e_1} \left(k\, e_k + (n-k) e_k \right) \\ &= \frac{n-k}{n} \cdot \frac{e_{k+1}}{e_1} + \frac{k}{n} \cdot \frac{e_k}{e_1} \end{aligned}$$

■

In order to prove that the average fitness of populations in an evolution converges to the maximum possible average fitness, we need to know the effect of W on the fraction e_2/e_1. Therefore we prove the following.

Proposition 8
In the notation of Proposition 7 we have for each $k \in \{1, \ldots, n\}$

$$\frac{e'_k}{e'_{k-1}} \geq \frac{e_k}{e_{k-1}}$$

where $0/0 := 0$ and $e_0 := 1$.

To prove this we need an application of the Cauchy-Schwartz inequality.

Lemma 9
Let α, a_i, b_i and c_i be non-negative reals for $i \in \{1, \ldots, n\}$. Suppose that for every i, $a_i/b_i \geq \alpha b_i/c_i$. Then

$$\frac{\sum_{i=1}^n a_i}{\sum_{i=1}^n b_i} \geq \alpha \frac{\sum_{i=1}^n b_i}{\sum_{i=1}^n c_i}$$

Proof: Set x_i to $\sqrt{a_i/\alpha}$ and y_i to $\sqrt{c_i/\alpha}$ and apply the Cauchy-Schwartz inequality. ■

Proof: Of Proposition 8. Using Proposition 7 we rewrite the left hand side of the desired inequality such that it changes to

$$\frac{(n-k)e_{k+1} + ke_k}{(n-k+1)e_k + (k-1)e_{k-1}} \geq \frac{e_k}{e_{k-1}}$$

If $e_k = 0$ then the inequality holds for sure, otherwise we rearrange it to

$$(n-k)\frac{e_{k+1}}{e_k} + 1 \geq (n-k+1)\frac{e_k}{e_{k-1}}$$

We will prove the inequality with one subtracted from the left hand side. Rewriting this in terms of the histogram representation a of p and dividing out the common $k!$ from all the sums yields

$$\frac{\sum_{j=k+1}^n (j)_{k+1} a_j}{\sum_{j=k}^n (j)_k a_j} \geq \frac{\sum_{j=k}^n (j)_k a_j}{\sum_{j=k-1}^n (j)_{k-1} a_j}$$

where $(j)_l$ is the falling factorial defined as $(j)_l := \Pi_{i=j-l+1}^{j} i = j!/(j-l)!$. Note that, because $(j)_k$ is zero when $j < k$, we can let every sum range from 1 to n. This gives us an inequality that matches the requirements of Lemma 9, so the inequality holds. ■

This proposition can be used to prove the following inequality for the change of the fraction e_2/e_1 after an application of the operator $\mathcal{F}$ on a population. Call a population *non-zero* if the probability p_1 of the binary string consisting of all 0's is strictly less than unity.

Lemma 10
Let e be the vector representation of a non-zero population p. Denote by e^* the vector representation of $\mathcal{F}(p)$. Then

$$\frac{e_2^*}{e_1^*} \geq \frac{2n-1}{2n} \cdot \frac{e_2}{e_1} + \frac{1}{2n}$$

Proof: For the proof we will also need the vector representation e' of $W(p)$. Then we can write

$$\begin{aligned}
\frac{e_2^*}{e_1^*} &= \frac{e_2' + {e_1'}^2}{2e_1'} \\
&= \frac{1}{2}e_1' + \frac{1}{2} \cdot \frac{e_2'}{e_1'} \\
&\geq \frac{1}{2}e_1' + \frac{1}{2} \cdot \frac{e_2}{e_1} \\
&= \frac{1}{2}\left(\frac{n-1}{n} \cdot \frac{e_2}{e_1} + \frac{1}{n}\right) + \frac{1}{2} \cdot \frac{e_2}{e_1} \\
&= \frac{2n-1}{2n} \cdot \frac{e_2}{e_1} + \frac{1}{2n}
\end{aligned}$$

■

We will now prove that starting with a zero-free population, the evolution will converge to a population with the maximum average fitness value n. Where *zero-free* means that the probability p_1 of the binary string consisting of only 0's is zero (remember that we identified 1 with the string of length n consisting of all zeros). That is, we will show that $\lim_{n\to\infty} n - \mathbf{Av}(p(n)) = 0$ provided that $(p(0))_1 = 0$.

Theorem 11
For every zero-free population p

$$n - \mathbf{Av}(W(M(p))) \leq (n - \mathbf{Av}(p))\left(1 - \frac{1}{2n}\right)$$

Proof: Because p is zero-free, there exists a population q such that $W(q) = p$. Let e, e', e^* and e^+ be the vector representations of q, $W(q) = p$, $\mathcal{F}(q)$ and $W(\mathcal{F}(q)) = W(M(p))$ respectively. Since $n - \mathbf{Av}(p)$ equals $n(1 - e_1')$, it is sufficient to prove the corresponding inequality for the e_1''s.

$$\begin{aligned}
1 - e_1^+ &= \frac{n-1}{n}\left(1 - \frac{e_2^*}{e_1^*}\right) \\
&\overset{(1)}{\leq} \frac{n-1}{n}\left(1 - \frac{e_2}{e_1}\right)\left(1 - \frac{1}{2n}\right) \\
&= (1 - e_1')\left(1 - \frac{1}{2n}\right)
\end{aligned}$$

Where (1) is an application of Lemma 10. Multiplication of both sides of the inequality with n concludes the proof. ■

The reproduction operator does not change the average fitness, as shown in Theorem 6. So we have the following lower bound on the average fitness of the populations after r generations.

Corollary 12

$$\begin{aligned}\mathbf{Av}(p_r) &\geq n - (n - \mathbf{Av}(p_1)) \left(1 - \frac{1}{2n}\right)^{r-1} \\ &\geq n - (n - \mathbf{Av}(p_1)) \exp\left(-\frac{r-1}{2n}\right)\end{aligned}$$

There are initial populations (called normal populations) for which the average fitness does not increase much faster than the lower bound. Normal populations are populations that have a decreasing sequence of expectation fractions e_m/e_{m-1} for increasing m.

Definition 7
A population is called normal if

$$e_0 \geq \frac{e_1}{e_0} \geq \frac{e_2}{e_1} \geq \cdots \geq \frac{e_n}{e_{n-1}}$$

where $0/0$ is regarded as 0 and $e_0 := 1$.

An example of a normal population is the uniform distribution on singletons, which has the vector representation $(1/n, 0, 0, \ldots, 0)$. Normality is preserved under M and W of which we omit the proof. The difference between the maximal fitness and the average fitness in between the generations will not increase faster than with a factor $1 - 1/n$. To complement Lemma 10 we will prove the following inequality.

Lemma 13
Using the notation of Lemma 10 with the additional condition that the population p is normal,

$$\frac{e_2^*}{e_1^*} \leq \frac{n-1}{n} \cdot \frac{e_2}{e_1} + \frac{1}{n}$$

Proof:

$$\begin{aligned}\frac{e_2^*}{e_1^*} &= \frac{e_2' + {e_1'}^2}{2e_1'} \\ &= \frac{1}{2}e_1' + \frac{1}{2} \cdot \frac{e_2'}{e_1'} \\ &\leq \frac{1}{2}e_1' + \frac{1}{2}e_1' = e_1' \\ &= \frac{n-1}{n} \cdot \frac{e_2}{e_1} + \frac{1}{n}\end{aligned}$$

■

With this we can prove that starting with a normal population, the evolution converges relatively slowly.

Theorem 14
Let p be some non-zero normal population. Then

$$n - \mathbf{Av}(W(M(p))) \geq (n - \mathbf{Av}(p)) \left(1 - \frac{1}{n}\right)$$

Proof: Using the same notation as in the proof of Theorem 11 we get

$$\begin{aligned} 1 - e_1^+ &= \frac{n-1}{n}\left(1 - \frac{e_2^*}{e_1^*}\right) \\ &\geq \frac{n-1}{n}\left(1 - \frac{e_1}{e_2}\right)\left(1 - \frac{1}{n}\right) \\ &= (1 - e_1')\left(1 - \frac{1}{n}\right) \end{aligned}$$

Multiplication of both sides of this inequality by n gives the result. ■

This result can again be expressed in average fitness. So we have the following upper bound on convergence.

Corollary 15
Suppose that p_0 is a normal population. Then

$$\mathbf{Av}(p_r) \leq n - (n - \mathbf{Av}(p_1)) \left(1 - \frac{1}{n}\right)^{r-1}$$

5. Genetic Programming

In genetic programming an individual is a tree representing a function, which is to be optimized. An example of an individual is illustrated in Figure 2.3. Each internal node of the individual is a mathematical operator that takes as arguments the values of its subtrees and passes its output to its parent node in the direction of the root of the tree. The arguments of the function are input at the leaves of the tree. The output of the function computed by a tree is found at the root of the tree.

The child operator is illustrated in Figure 2.4. A child is constructed by selecting a node in each copy of the parents, removing the subtree rooted at the selected node in the mother and replacing it by the subtree rooted at the node selected in the father.

Genetic programming solves the representation problem of genetic algorithms by choosing a tree to represent a function, which is a natural way to represent a function. Genetic programming does not solve the other problem of genetic algorithms, namely: How to choose an operator set? This requires knowledge of the art of the solution to the problem. This is the major drawback of genetic programming.

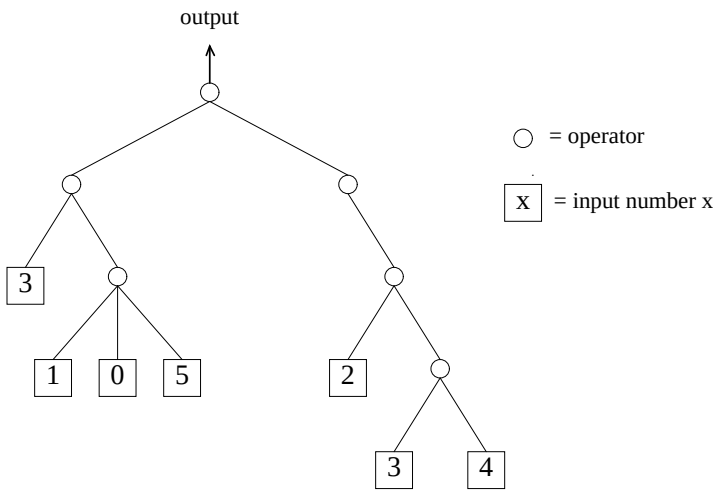

Figure 2.3: Example of an individual in genetic programming

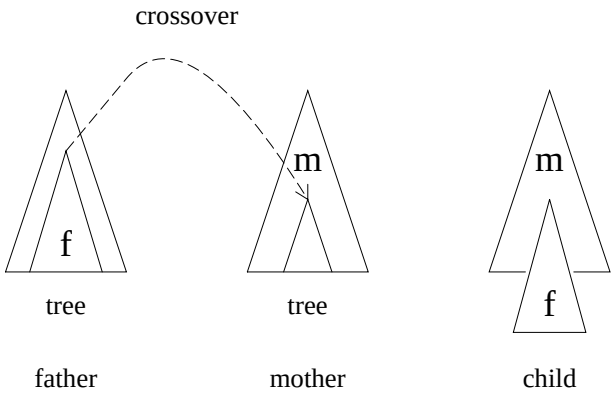

Figure 2.4: The child operator in genetic programming

5.1 Choosing an operator set

To illustrate the problem of choosing an operator set, mentioned in the previous section, we give some examples of operator sets from [15].

Example 1

Cart centering: Find a function that will bring a cart moving along a track to rest at a designated target point in minimal time. The function has as input the current position $x(t)$ and velocity $v(t)$ of a cart at time t and as output a force f, which is applied to the cart.

The chosen operator set: $\{+, -, *, /, |\,|, >\}$.

Example 2

Simple symbolic regression: Find a function that will closely fit a given finite sample of data. The function has as input the independent variable x and as output the dependent variable y.

The chosen operator set: $\{+, -, *, \sin, \cos, \exp, \log\}$.

Example 3

Boolean multiplexer: Find a function that performs the boolean 11-multiplexer function. The function has as input the three address and eight data bits and as output the data bit singled out by the address bits.

The chosen operator set: $\{\wedge, \vee, \neg, \textbf{if}\}$, where **if** takes three arguments a, b and c and is equal to $(a \wedge b) \vee (\neg a \wedge c)$.

From the three examples above it should be clear that choosing the operator set in genetic programming is by no means a trivial matter. It does require knowledge of the type of solution to the problem.

Chapter 3
Finite Automaton Learning System

To bridge the gap between machine learning and evolutionary systems we propose a system that uses a slightly extended form of genetic programming that, given the behavior of a finite automaton, tries to find a small digital circuit that is functionally equivalent to the given automaton. We chose to take automata as the class of tasks to learn because learning was supposed to be about the acquisition of programs, *i.e.*, descriptions that can be interpreted step by step. The next step would be to look at – programs for – Turing machines, but we decided to look at finite automaton first. Fortunately it is not necessary to change very much in the genetic programming approach to evolve – and consequently learn – finite automata, because a finite automaton is fully specified by its state transition function. In order to use genetic programming we need a representation of this state transition function in the form of an operator tree. We achieved this by representing the states as bit strings. This allows us to view the state transition function as a sequence of boolean functions, one for each bit of the state variable. Each boolean function can be represented as an operator tree using standard boolean connectives. This results in a forest of operator trees rather than a single tree, but this does not change the method significantly.

We set up the Finite Automaton Learning System in such a way that it is fairly easy to implement numerous variations on the theme of genetic programming. The idea behind this is that we want to investigate for what circumstances some variations or parameter settings work better than others, so we want to be able to try them all (in principle). We will now give a description of FALS in the terminology of genetic algorithms and genetic programming and where appropriate we will point out various alternatives where we did not decide to implement it one way or another.

Each evolution starts with a population which is called the *start population* drawn from a suitable distribution over all possible individuals. We can not use the uniform distribution because there are countably many individuals, so we opted for a distribution that favors small individuals, drawing without replacement to avoid many inclusions of the same individual.

In contrast to the evolutionary systems from the previous chapter, the fitness of an individual is not based on verification of output on single inputs, but on verification of output on an arbitrary long sequence of inputs. That is, a sequence of inputs is generated and fed one symbol at a time to the individual. After each symbol the new state of the individual specifies a new output symbol that is compared to the desired output symbol (the symbol that the target finite automaton would have produced if it had been presented with the same sequence of inputs so far). Obviously the closer the individual approaches the output given by the environment, the fitter the individual should be. For example, fitness could be measured as the number of output values of the individual exactly matching the corresponding output of the environment.

After the fitness of the individuals in the population is determined, individuals are selected according to fitness to serve as input to operators that produce the individuals for the next generation. Before we go into a description of the genetic operators that can be used in FALS, we will first have a closer look at the representation of an individual.

1. The individual

An individual in FALS as depicted in Figure 3.1 consists of a forest of m trees, a state memory, an input memory and an output memory. The three memory parts are each built out of a number of boolean variables called bits, m state variables, n input variables and p output variables. Each tree of the forest is an operator tree representing a boolean function. In the current implementation we restricted the operator set to contain only the binary NAND operator. This restriction is not fundamental in the sense that every possible boolean functions can be represented using only binary NAND operators. So each internal node in the tree represents a NAND gate. Every leaf of the tree is a reference to either a bit in state memory or a bit in input memory. There is a one-to-one correspondence between trees and state variables, every tree is used to calculate the next value of the corresponding state variable. The interpretation of the memory parts and the trees as a finite automaton is as follows. At any time t the input memory of an individual contains a binary representation of the input symbol at time t, the state memory contains a binary representation of the state at time t of the automaton that the individuals represents and at time $t+1$ the output contains a binary representation of the output symbol that the individual produced having read the input symbols up to time t.

In FALS a tree represents a boolean formula, where we take a boolean formula to be a boolean circuit, whose underlying structure is a tree As an example a binary tree of small size is illustrated in Figure 3.2. Each internal node in the tree is a binary NAND operator. The result of the formula is found at the root of the tree. Each internal node in a tree has as arguments the boolean values of its subtrees and passes its computed value in the direction of the root of the tree. The leaves of the tree are fed the boolean arguments of the function. The leaves obtain their boolean values by copying the value of the variable they refer to. Each leaf can refer to a variable in state memory or a variable in input memory.

The state transition function of an individual is represented by a sequence of trees called a forest. Because each tree in the forest represents a boolean formula, a forest represents a

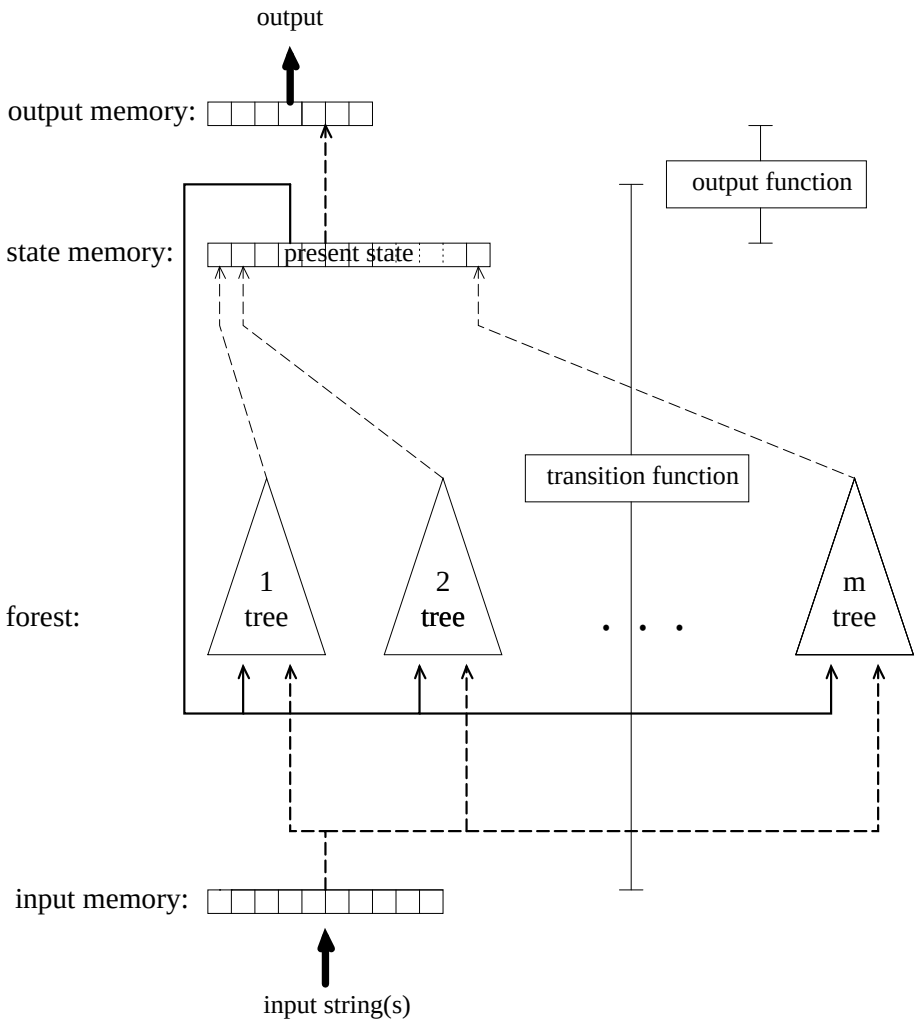

Figure 3.1: An individual in FALS

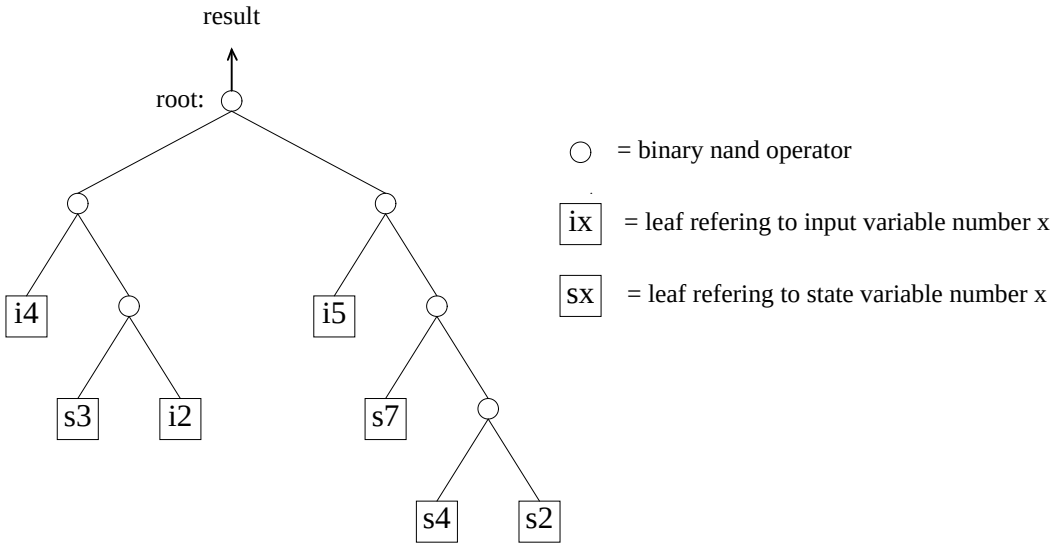

Figure 3.2: Example of a tree in a FALS individual

boolean circuit. Arguments to the forest are thus boolean values in state memory and input memory. The result of the forest is the sequence of results of the trees making up the forest. The result of the forest is copied in state memory, where the result of each tree is stored in the state variable with which it is associated.

To determine the fitness of an individual, it is run through a *test*. For each test in which the individual is compared to the environment, the environment generates a sequence of input symbols. These are translated into bit strings and fed to the individuals, one bit string at a time. Each bit string is used together with the current state bits to compute the next values of the state bits. These new values determine the new values of the output memory that are interpreted as a new output symbol. This symbol is then compared to the output symbol that the target finite automaton produced on the same sequence of inputs. The output function that maps the state bits onto output bits is a fixed function mapping the first p boolean values of state memory to output memory. To make this possible the number of state variables, which can vary, is bounded from below by the number of output variables, *i.e.*, $m \geq p$.

Before providing the first input symbol to the individual the state bits are all set to false, so this all-false state corresponds to the initial state of the finite automaton.

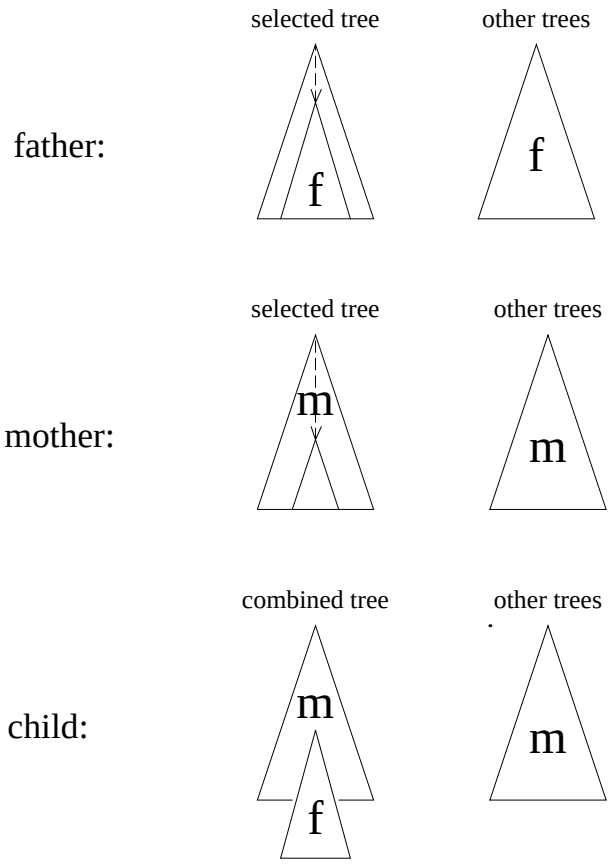

Figure 3.3: The child operator in FALS

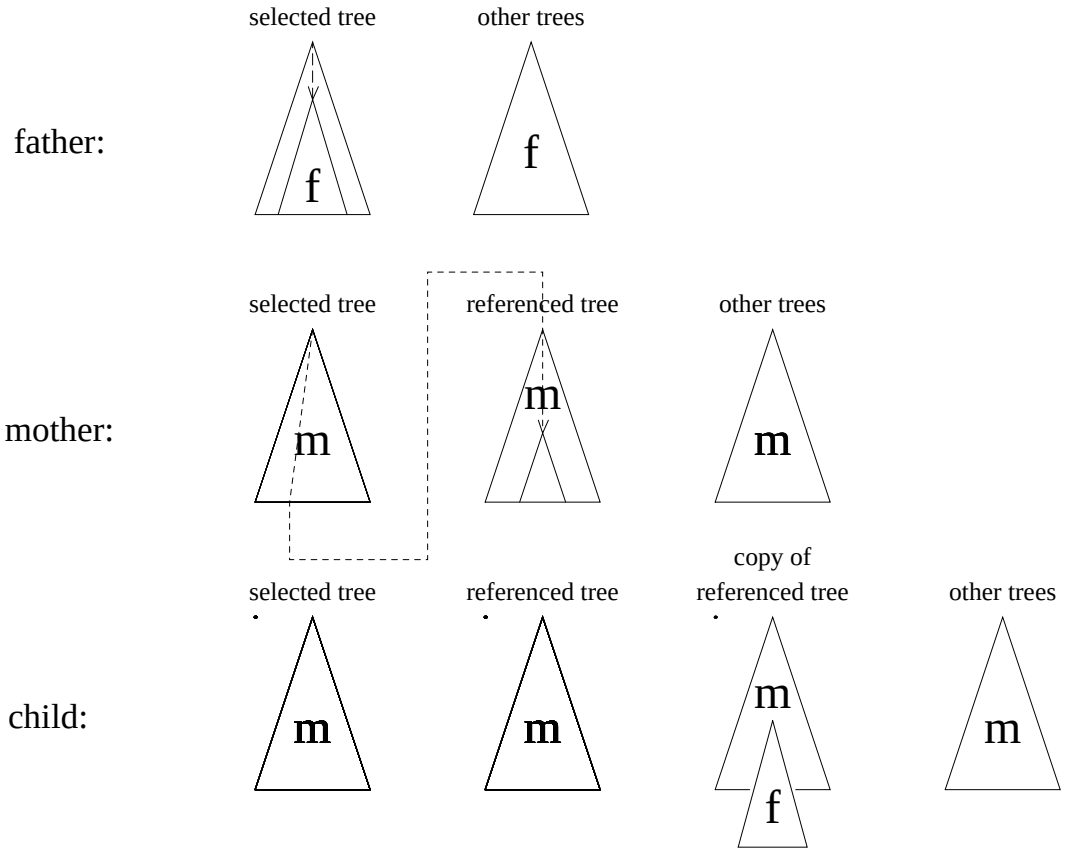

Figure 3.4: Increasing the number of state variables

2. Genetic operators

The genetic operator 'child' is derived from the one used in genetic programming. The child operation is illustrated in Figure 3.3. First a node is chosen from a selected tree of the father. The subtree rooted at this node is copied. Next a copy of the mother is made. In the copy of the mother a node is chosen from a selected tree. The subtree rooted at this node is replaced by the subtree copied from the father. The resulting individual is the child. At present selection of nodes and trees in FALS is done at random, using one of a variety of methods that specify the probability that a node is chosen.

Since it is not known in advance how many states an individual will require to adapt to the particular environment it is exposed to, we want the child operator to change the number of state bits in some circumstances. This is done by taking a special action in case a leaf referring to a state bit is selected in the mother. In this case the state variable and its associated tree are copied and added to the individual as the last tree in the forest, increasing the size of the forest by one. This process is illustrated in Figure 3.4 (in the figure only the trees are shown, so imagine a state variable at the root of each tree). In this extra tree a node is selected as

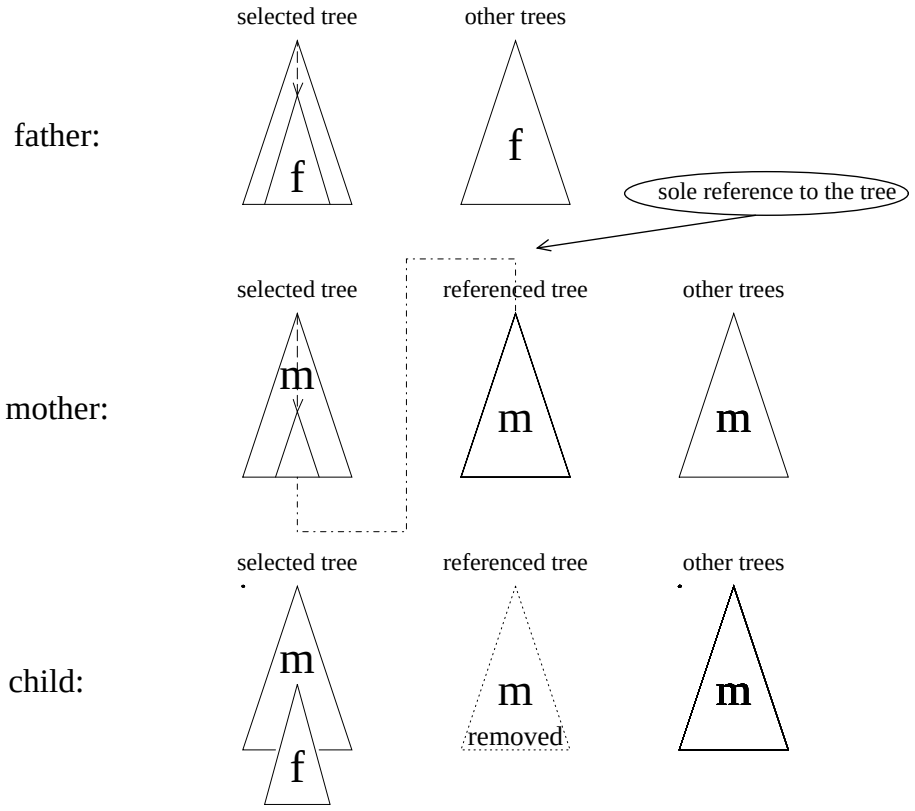

Figure 3.5: Decreasing the number of state variables

if it were the first node selected. The rest of the child operation remains unaltered. Realize though that it is possible that several leaves in a row are selected, each of which results in the addition of a state variable and an associated tree. This method of increasing the number of variables has the benefit that it allows the boolean functionality of leaves referring to the same state variable to be independently modified.

To decrease the number of variables, we implemented a simple form of garbage collection. If a state variable is no longer related to output then it is removed from the individual together with its associated tree. So a state variable that is used by the output function is never removed. Other state variables that are not related via some chain of references among trees in the forest to a state bit used by the output function, are removed. The simplest case in which a state variable is removed is illustrated in Figure 3.5 (in the figure only the trees are shown, so imagine a state variable at the root of each tree). In this case a state variable looses its sole reference.

3. Circuit and architecture design

FALS is aimed at the circuit design problem. In a theoretical sense the problem of circuit design is hard, in terms of complexity theory it is NP-hard. In a practical sense the problem of circuit design is even harder. In todays micro electronics the number of components on a micro chip is so large that it is necessary to start with the choice of an architecture, being a high-level decision on the design of a circuit. As a consequence one not only faces the problem of circuit design, but also the problem of choosing an architecture, like for example the random access machine architecture, whenever one wants to build a circuit or machine of some complexity.

Apart from these two problems in circuit design there are all kinds of restrictions put upon circuits by the production process. These restrictions involve the size of components, their layout on a chip, the number of wires connecting components and trade offs between chip area, computation delay, power or energy consumption, etc.

FALS is well suited to find solutions to just the circuit design problems. The following general description of the circuit design problem is meant to make this clear. Stated in the most general way the circuit design problem receives as input a description of the functional behavior of a circuit and produces the *best* circuit complying with this specification. In FALS the environment holds the description of the functional behavior of a circuit and the fittest individual is the 'best' circuit found so far. It is important to realize that evolutionary systems are a good method for optimization of problems in NP. That is they can find sub-optimal 'solutions' to optimization problems. So with FALS we hope to find circuits that are among the best complying with the specification of a circuit as represented by the environment. We are not aiming at finding the best circuit complying with the specification, since this is an NP-hard problem, which in practice means that we can not solve it.

The tendency to minimize circuits is achieved by subtracting a penalty from the fitness of individuals that is proportional to their size. We are still researching various ways to do this. In the current implementation the penalty function apparently increases too fast with the size of the individual, because the system stabilizes on one-state circuits for non-trivial reference automata. The problem is that normal fitness is determined by the number of mistakes that the individual makes. To compare this to the size of the individual is like comparing apples to oranges. In the near future we want to find out if this can be solved using the minimum description length principle (MDL) [18]. The idea is that the cost associated with an individual is the number of bits needed to describe the individual plus the number of bits needed to correct its output.

REFERENCES

1. M. Anthony, N. Biggs, *Computational Learning Theory*, Cambridge Univ. Press, 1992.
2. R. Bayer, *Symmetric binary B-trees: Data structure and maintenance algorithms*, Acta Informatica, 1972, 1:290-306.
3. S.A. Cook, R.A. Reckhow, *Time bounded random access machines*, J. Comput. Syst. Sci. 7 (1973), 354-375.
4. S.A. Cook, *Towards a Complexity Theory of Synchronous Parallel Computation*, Einseign. Math. 27 (1981), 99-124.
5. T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.
6. C. Darwin, *On the Origin of Species by Means of Natural Selection*, John Murray, 1859.
7. P.W. Dymond, *Simultaneous Resource Bounds and Parallel Computation*, Ph.D. thesis, Univ. of Toronto, Dept. of Comp. Sc., 1980.
8. H.H. Ehrenburg, *FALS, Finite Automaton Learning System*, Masters thesis, Univ. of Amsterdam, Dept. of Comp. Sc., 1993.
9. H.H. Ehrenburg, H.A.N. van Maanen, *A finite automaton learning system using genetic programming*, Proc. 4th Belgian-Dutch Conference on Machine Learning, Erasmus University Rotterdam, Dept. of Comp. Sc., 1994.
10. D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1989.
11. J.H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*, Univ. of Michigan Press, 1975.
12. J.E. Hopcroft, J.D. Ullman, *Introduction to Automata theory, Languages, and Computation*, Addison-Wesley, 1979.
13. M. Kearns, M. Li, L. Pitt, L. Valiant, *On the learnability of Boolean Formulae*, Proc. 19th ACM Symp. Theory of Computing, 1987, 285-295.
14. S.C. Kleene, *Representation of events in nerve nets and finite automata, in: C. Shannon and J. McCarthy, eds.*, Automata Studies, Princeton Univ. Press, Princeton, NJ, 1956, 3-41.
15. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.
16. K.B. Krohn, J.L. Rhodes, *Algebraic Theory of Machines*, Trans. Am. Math. Soc. 116, 1965, 450-464.
17. H.R. Lewis, C.H. Papadimitrou, *Elements of the theory of computation*, Prentice-Hall, 1981.
18. M. Li, P.M.B. Vitányi, *Learning simple concepts under simple distributions*, SIAM. J. Computing, 20:5(1991), 911-935.
19. M.L. Minsky, *Computation: Finite and infinite machines*, Prentice-Hall, 1967.

20. W.S. McCullough, W. Pitts, *A logical calculus of ideas immanent in nervous activity*, Bull. Math. Biophys. 5, 1943, 115-133.

21. L. Pitt, M. Warmuth, *The minimum consistent DFA problem cannot be approximated within any polynomial*, Proc. 21st ACM Symp. Theory of Computing, 1989, 421-432.

22. Y. Rabinovich, A. Sinclair, A. Widgerson, *Quadratic Dynamical Systems*, Proc. 33th Symp. on Foundations of Computer Science 1992, 304-313.

23. Y. Rabinovich, A. Widgerson, *An Analysis of a Simple Genetic Algorithm*, Proc. 4th Conf. on Genetic Algorithms, 1991, 215-221.

24. J.E. Savage, *The Complexity of Computing*, Wiley, 1976.

25. J. Shallit, *Automaticity: Properties of a Measure of Descriptional Complexity*, (rough draft of which abstract has been submitted to STACS '94)

26. H.M. Sheffer, *A set of Five Independent Postulates for Boolean Algebras, with Application to Logical Constants*, Am. Math. Soc. Tr. nr. 14, 1913,

27. R.J. Solomonoff, *A formal theory of inductive inference*, Information and Control 7(1964), 1-22, 224-254.

28. R.J. Solomonoff, *Complexity-based induction systems: comparisons and convergence theorems*, IEEE Trans. Inform. Th. IT-24(1978), 422-432.

29. L.G. Valiant, *A Theory of the Learnable*, Comm. ACM, 27 (1984), 1134-1142.

30. A.K. Zvonkin, L.A. Levin, *The complexity of finite objects and the development of concepts of information and randomness by means of the theory of algorithms*, Russ. Math. Surv., 25:6(1970), 83-124.